Parallel Factorizations with Algorithmic Blocking

Jaeyoung Choi

School of Computing, Soongsil University, Seoul, KOREA

Abstract. Matrix factorization algorithms such as LU, QR, and Cholesky, are the most widely used methods for solving dense linear systems of equations, and have been extensively studied and implemented on vector and parallel computers. In this paper, we present parallel LU, QR, and Cholesky factorization routines with an "algorithmic blocking" on 2-dimensional block cyclic data distribution. With the algorithmic blocking, it is possible to obtain the near optimal performance irrespective of the physical block size. The routines are implemented on the SGI/Cray T3E and compared with the corresponding ScaLAPACK factorization routines.

1 Introduction

In many linear algebra algorithms the distribution of work may become uneven as the algorithm proceeds, for example as in LU factorization algorithm [7], in which rows and columns are successively eliminated from the computation. The way in which a matrix is distributed over the processors has a major impact on the load balance and communication characteristics of a parallel algorithm, and hence largely determines its performance and scalability.

The two-dimensional block cyclic data distribution [9], in which matrix blocks separated by a fixed stride in the row and column directions are assigned to the same processor, has been used as a general purpose basic data distribution for parallel linear algebra software libraries because of its scalability and load balance properties. And most of the parallel version of algorithms have been implemented on the two-dimensional block cyclic data distribution [5,13].

Since parallel computers have different performance ratios of computation and communication, the optimal computational block sizes are different from one another to generate the maximum performance of an algorithm. The data matrix should be distributed with the machine specific optimal block size before the computation. Too small or large a block size makes getting good performance on a machine nearly impossible. In such case, getting a better performance may require a complete redistribution of the data matrix.

The matrix multiplication, $\mathbf{C} \Leftarrow \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$, might be the most fundamental operation in linear algebra. Several parallel matrix multiplication algorithms have been proposed on the two-dimensional block-cyclic data distribution [1,6, 8,12]. High performance, scalability, and simplicity of the parallel matrix multiplication schemes using rank-K updates has been demonstrated [1,12]. It is

V.N. Alexandrov et al. (Eds.): ICCS 2001, LNCS 2073, pp. 802–811, 2001.

assumed that the data matrices are distributed on the two-dimensional block cyclic data distribution and the column block size of **A** and the row block size of **B** are K. However getting a good performance when the block size is very small or very large is difficult, since the computation are not effectively overlapped with the communication. The LCM (Least Common Multiple) concept has been introduced to DIMMA [6] to use a computationally optimal block size irrespective of the physically distributed block size for the parallel matrix multiplication. In DIMMA, if the physical block size is smaller than the optimal block size, the small blocks are combined into a larger block. And if the physical block size is larger than the optimal block size, the block is divided into smaller pieces. This is the "algorithmic blocking" strategy.

There have been several efforts to develop parallel factorization algorithms with the algorithmic blocking on distributed-memory concurrent computers. Lichtenstein and Johnsson [11] developed and implemented block-cyclic order elimination algorithms for LU and QR factorization on the Connection Machine CM-200. They used a cyclic order elimination on a block data distribution, the only scheme that the Connection Machine system compilers supported.

P. Bangalore [3] has tried to develop a data distribution-independent LU factorization algorithm. He recomposed computational panels to obtain a computationally optimal block size, but followed the original matrix ordering. According to the results, the performance is superior to the other case, in which the matrix is redistributed when the block size is very small. He used a tree-type communication scheme to make computational panels from several columns of processors. However, using a pipelined communication scheme, if possible, which overlaps communication and computation effectively, would be more efficient.

The actual algorithm which is selected at runtime depending on input data and machine parameters is called "polyalgorithms" [4]. We are developing "PoLAPACK" (Poly LAPACK) factorization routines, in which computers select the optimal block size at run time according to machine characteristics and size of data matrix. In this paper, we expanded and generalized the idea in [11]. We developed and implemented parallel LU, QR, and Cholesky factorization routines with the algorithmic blocking on the 2-dimensional block cyclic data distribution. With PoLAPACK, it is always possible to have the near optimal performance of LU, QR, and Cholesky factorization routines on distributed-memory computers irrespective of the physical data-distribution on distributed-memory concurrent computers if all of the processors have the same size of submatrices.

The PoLAPACK LU, QR, and Cholesky factorization routines are implemented on the SGI/Cray T3E at KISTI Supercomputing Center, Korea. And their performance is compared with that of the corresponding ScaLAPACK factorization routines.

2 PoLAPACK LU Factorization Algorithm

The basic LU factorization routine is to find the solution vector x after applying LU factorization to A from the following linear equation, $A\,x = b$. After

converting A to $P \cdot A = L \cdot U$, compute y from $L\,y = b_0$, where $U \cdot x = y$ and $P \cdot b = b_0$. And compute x.

Most of the LU factorization algorithms including LAPACK [2] and ScaLAPACK [7] find the solution vector x after computing the factorization of $P \cdot A = L \cdot U$. And in the ScaLAPACK factorization routines, a column of processors performs a factorization on its own column of blocks, and broadcasts it to others. Then all of processors update the rest of the data matrix. The basic unit of the computation is the physical size of the block, with which the data matrix is already distributed over processors.

We measured the performance the ScaLAPACK LU factorization routine and its solution routine with various block sizes on the SGI/Cray T3E. Figure 1 shows the performance on an 8×8 processor grid from $N = 1,000$ to 20,000 with block sizes of $N_b = 1, 6, 24, 36$, and 60. It shows that the near optimal performance is obtained when $N_b = 60$, and almost the same but slightly slower when $N_b = 36$ or 24. The performance deteriorated by 40% when $N_b = 6$ and 85% when $N_b = 1$. If the data matrix is distributed with $N_b = 1$, it may be much more efficient to perform the factorization after redistributing the data matrix with the optimal block size.

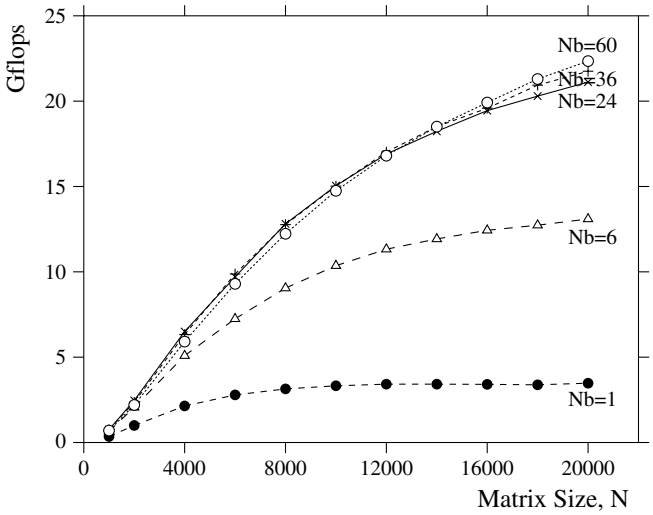


Fig. 1. Performance of ScaLAPACK LU factorization routine on an 8×8 SGI/Cray T3E

In ScaLAPACK, the performance of the algorithm is greatly affected by the block size. However the PoLAPACK LU factorization is implemented with the concept of algorithmic blocking and always shows the best performance of $N_{opt} = 60$ irrespective of physical block sizes.

If a data matrix A is decomposed over 2-dimensional $p \times q$ processors with the block cyclic data distribution, it may be possible to regard the matrix A being

decomposed along the row and column directions of processors. Then the new decomposition along the row and column directions are the same as applying permutation matrices from the left and the right, respectively. One step further. If we want to compute a matrix with a different block size, we may need to redistribute the matrix, and we can assume that the redistributed matrix is of the form $P_p \cdot A \cdot P_q^T$, where P_p and P_q are permutation matrices. It may be possible to avoid redistributing the matrix physically if the new computation doesn't follow the given ordering of the matrix A. That is, by assuming that the given matrix A is redistributed with a new optimal block size and the resulting matrix is $P_p \cdot A \cdot P_q^T$, it is now possible to apply the factorization to A with the optimal block size for the computation. And this factorization will show the same performance regardless of the physical block sizes if each processor gets the same size of the submatrix of A. These statements are illustrated with the following equations,

$$(P_p\, A\, P_q^T) \cdot (P_q x) \;=\; P_p \cdot b. \tag{1}$$

Let $A_1 = P_p\, A\, P_q^T$, and $x_1 = P_q\, x$. After factorizing $P_1 A_1 = P_1 \cdot (P_p A P_q^T) = L_1 \cdot U_1$, then we compute the solution vector x. The above equation Eq. 1 is transformed as follows:

$$L_1 \cdot U_1 \cdot (P_q x) \;=\; L_1 \cdot U_1 \cdot x_1 \;=\; P_1 \cdot (P_p b) \;=\; b_1.$$

Then, y_1 is computed from

$$L_1 \cdot y_1 \;=\; b_1, \tag{2}$$

and x_1 is computed from

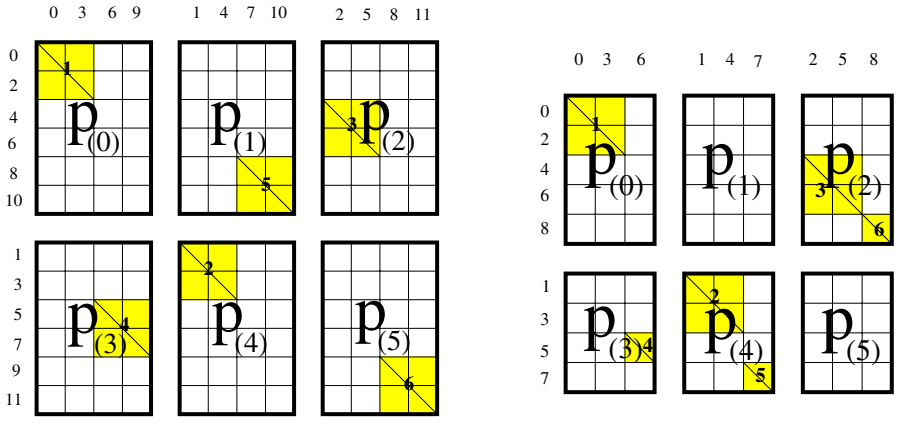
$$U_1 \cdot x_1 \;=\; y_1. \tag{3}$$

Finally the solution vector x is computed from

$$P_q \cdot x \;=\; x_1. \tag{4}$$

The computations are performed with A and b in place with the optimal block size, and x is computed with P_q as in Eq. 4. But we want $P_p \cdot x$ rather than x in order to make x have the same physical data distribution as b. That is, it is required to compute

$$P_p \cdot x \;=\; P_p \cdot P_q^T \cdot x_1. \tag{5}$$



(a) 12 x 12 blocks on 2 x 3 processes

(b) 9 x 9 blocks on 2 x 3 processes

Fig. 2. Computational Procedure in PoLAPACK. Matrices of 12×12 and 9×9 blocks are distributed on 2×3 processors with $N_{opt} = N_b$ and $N_{opt} = 2 \cdot N_b$, respectively.

3 Implementation of PoLAPACK LU Factorization

Figure 2 shows the computational procedure of the PoLAPACK LU factorization. It is assumed that a matrix A of 12×12 blocks is distributed over a 2×3 processor grid as in Figure 2(a), and the LU routine computes 2 blocks at a time (imagine $N_b = 4$ but $N_{opt} = 8$). Since the routine follows the 2-D block cyclic ordering, the positions of the diagonal blocks are regularly changed by incrementing one column and one row of processors at each step. However, if A is 9×9 blocks as in Figure 2(b), the next diagonal block of $A(5, 6)$ on $p_{(3)}$ is $A(7, 7)$ on $p_{(4)}$, not on $p_{(1)}$. Then the next block is $A(8, 8)$ on $p_{(2)}$. The computational procedure of the PoLAPACK is very complicated.

We implemented the Li and Coleman's algorithm [10] on a two dimensional processor grid for the PoLAPACK routines. But the implementation is much more complicated since the diagonal block may not be located regularly if p is not equal to q as in Figure 2.

Though p is equal to q, the implementation is still complicated. Figure 3(a) shows a snapshot of the Li and Coleman's algorithm from the processors point-of-view, where 9×9 blocks of an upper triangular matrix T are distributed over a 3×3 processor grid with $N_b = N_{opt} = 1$. Let's look over the details of the algorithm to solve $x = T \setminus b$.

At first, the last block at $p_{(8)}$ computes $x(9)$ from $T(9, 9)$ and $b(9)$. Processors in the last column update 2 blocks - actually $p_{(2)}$ and $p_{(5)}$ update $b(7)$ and $b(8)$, respectively - and send them to their left processors. The rest of b ($b(1 : 6)$) is updated later. At the second step, $p_{(4)}$ computes $x(8)$ from $T(8, 8)$ and $b(8)$, the latter is received from $p_{(5)}$. While $p_{(1)}$ receives $b(7)$ from $p_{(2)}$, updates it, and

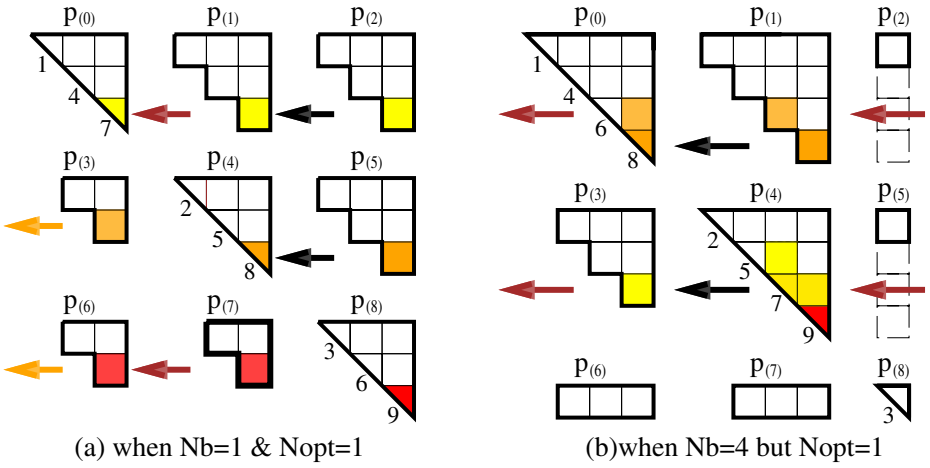


Fig. 3. A snapshot of PoLAPACK solver. A matrix T of 9×9 blocks is distributed on 3×3 processors with $N_b = 1$ and $N_b = 4$, respectively, while the optimal computational block size for both cases is $N_{opt} = 1$.

sends it to $p_{(0)}$, $p_{(7)}$ updates a temporal $b(6)$ and sends it to $p_{(6)}$.

Figure 3(b) shows the same size of the matrix distribution T with $N_b = 4$, but it is assumed that the matrix T is derived with an optimal block size $N_{opt} = 1$. So the solution routine has to solve the triangular equations of Eq. 2 and Eq. 3 with $N_{opt} = 1$. The first two rows and the first two columns of processors have 4 rows and 4 columns of T, respectively, while the last row and the last column have 1 row and 1 column, respectively. Since $N_{opt} = 1$, the computation starts from $p_{(4)}$, which computes $x(9)$. Then $p_{(1)}$ and $p_{(4)}$ update $b(8)$ and $b(7)$, respectively, and send them to their left. The rest of b ($b(1:6)$) is updated later. At the next step, $p_{(0)}$ computes $x(8)$ from $T(8,8)$ and $b(8)$, the latter is received from $p_{(1)}$. While $p_{(3)}$ receives $b(7)$ from $p_{(4)}$, updates it, and sends it to the left $p_{(5)}$, $p_{(0)}$ updates a temporal $b(6)$ and sends it to its left $p_{(2)}$. However $p_{(2)}$ and $p_{(5)}$ don't have their own data to update or compute at the current step, and hand them over to their left without touching the data. The PoLAPACK solver has to comply with this kind of all abnormal cases.

It may be necessary to redistribute the solution vector x to $P_p \cdot P_q^T \cdot x$ as in Eq. 5. However, if p is equal to q, then P_p becomes P_q, and $P_p \cdot P_q^T \cdot x = x$, therefore, the redistribution is not necessary. But if p is not equal to q, the redistribution of x is required to get the solution with the same data distribution as the right hand vector b. And if p and q are relatively prime, then the problem is changed to all-to-all personalized communication.

Figure 4 shows a case of the physical block size $N_b = 1$ and the optimal block size $N_{opt} = 2$ on a 2×3 processor grid. Originally the vector b is distributed with $N_b = 1$ as the ordering on the left of Figure 4. But the solution vector x

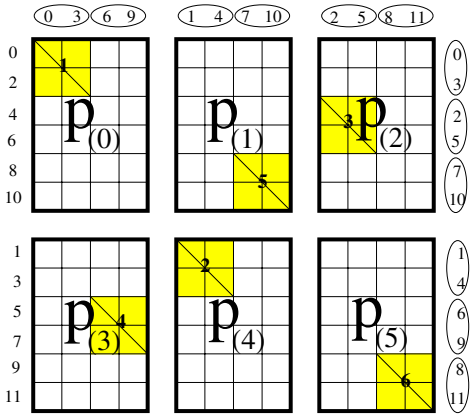


Fig. 4. A snapshot of PoLAPACK solver. A matrix T of 9×9 blocks is distributed on 3×3 processors with $N_b = 1$ and $N_b = 4$, respectively, while the optimal computational block size for both cases is $N_{opt} = 1$.

is distributed as the ordering on the right after the computation with $N_{opt} = 2$. The result is the same as a vector on the left is transposed twice – at first transposed with $N_b = 1$ to the vector on the top, then later transposed with $N_{opt} = 2$ to the vector on the right.

We implemented the PoLAPACK LU factorization routine and measured its performance on an 8×8 processor grid of the SGI/Cray T3E. Figure 5 shows the performance of the routine with the physical block sizes of $N_b = 1, 6, 24, 36,$ and 60, but the optimal block size of $N_{opt} = 60$. The performance lines are very close to the others and always show nearly the maximum performance irrespective of the physical block sizes. Since all processors don't have the same size of the submatrices of A with various block sizes, some processors have more data to compute than others, which causes computational load imbalance among processors and slight performance degradation.

4 PoLAPACK QR and Cholesky Factorization

The PoLAPACK QR factorization and its solution of the factored matrix equations are performed in a manner analogous to the PoLAPACK LU factorization and the solution of the triangular systems.

Figure 6 shows the performance of the ScaLAPACK and PoLAPACK QR factorizations and their solution on an 8×8 processor grid of the SGI/Cray T3E. Performance of the ScaLAPACK QR factorization routine depends on the physical block size, and the best performance is obtained when $N_b = 24$ on a SGI/Cray T3E. However the PoLAPACK QR factorization routine, which computes with the optimal block size of N_{opt}, always shows nearly the maximum performance independent of physical block sizes.

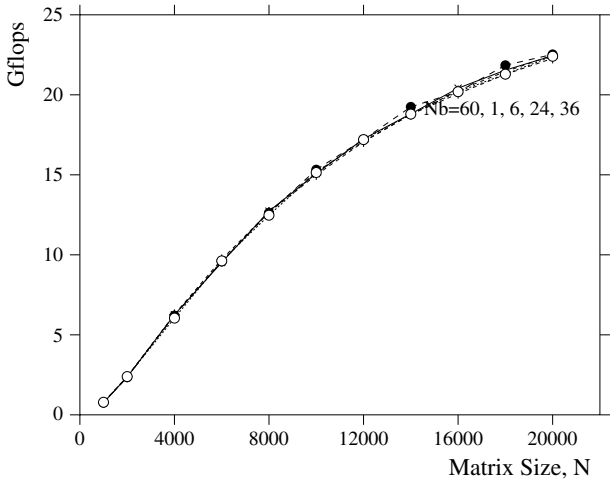


Fig. 5. Performance of PoLAPACK LU on an 8×8 SGI/Cray T3E

The Cholesky factorization factors an $N \times N$, symmetric, positive-definite matrix A into the product of a lower triangular matrix L and its transpose, i.e., $A = LL^T$ (or $A = U^T U$, where U is upper triangular).

Though A is symmetric, $P_p A P_q^T$ is not symmetric if $p \neq q$. That is, if $P_p A P_q^T$ is not symmetric, it is impossible to exploit the algorithmic blocking technique to the Cholesky factorization routine as used in the PoLAPACK LU and QR factorization. If $p \neq q$, the PoLAPACK Cholesky computes the factorization with the physical block size. That is, it computes the factorization as the same way of the ScaLAPACK Cholesky routine. However, it is possible to obtain the benefit of algorithmic blocking for the limited case of $p = q$.

Figure 7 shows the performance of the ScaLAPACK and the PoLAPACK Cholesky factorization and their solution on an 8×8 processor grid of the SGI/Cray T3E. Similarly, the performance of the ScaLAPACK Cholesky factorization routine depends on the physical block size. However the PoLAPACK Cholesky factorization routine, which computes with the optimal block size of $N_{opt} = 60$, always shows the maximum performance.

5 Conclusions

Generally in most parallel factorization algorithms, a column of processors performs the factorization on a column of blocks of A at a time, whose block size is already fixed, and then the other processors update the rest of the matrix. If the block size is very small or very large, then the processors can't show their optimal performance, and the data matrix may be redistributed for a better performance. The computation follows the original ordering of the matrix.

It may be faster and more efficient to perform the computation, if possible, by combining several columns of blocks if the block size is small, or by splitting

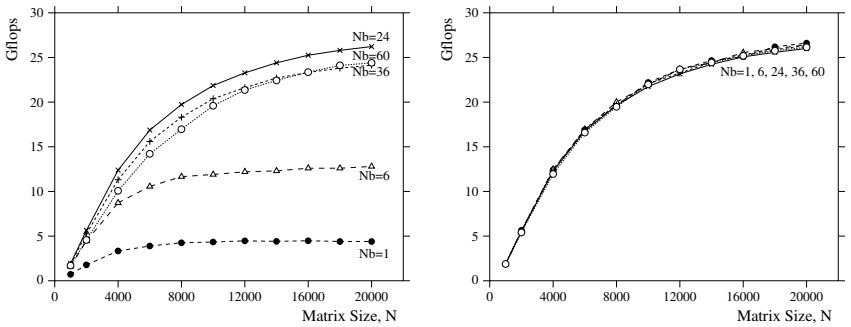


Fig. 6. Performance of ScaLAPACK QR and PoLAPACK QR on an 8×8 SGI/Cray T3E

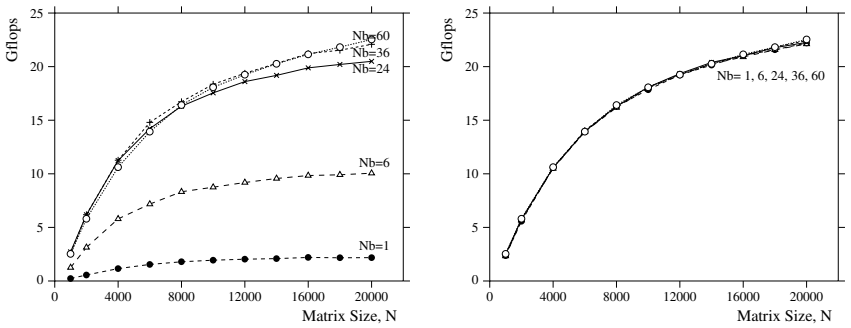


Fig. 7. Performance of ScaLAPACK and PoLAPACK Cholesky on an 8×8 SGI/Cray T3E

a large column of blocks if the block size is large. This is the main concept of algorithmic blocking. The PoLAPACK factorization routines rearrange the ordering of the computation. They compute $P_p A P_q^T$ instead of directly computing A. The computation proceeds with the optimal block size without physically redistributing A. And the solution vector x is computed by solving triangular systems, then converting x to $P_p P_q^T x$. The final rearrangement of the solution vector can be omitted if $p = q$ or $N_b = N_{opt}$.

According to the results of the ScaLAPACK and the PoLAPACK LU, QR, and Cholesky factorization routines on the SGI/Cray T3E, the ScaLAPACK factorization routines have a large performance difference with different values of N_b, however the PoLAPACK factorizations always show a steady performance, which is the near optimal, irrespective of the values of N_b. The routines we presented in this paper are developed based on the block cyclic data distribution. This simple idea can be easily applied to the other data distributions. But it is required to develop specific algorithms to rearrange the solution vector for each distribution.

References

1. R. C. Agarwal, F. G. Gustavson, and M. Zubair. A High-Performance Matrix-Multiplication Algorithm on a Distributed-Memory Parallel Computer Using Overlapped Communication. *IBM Journal of Research and Development*, 38(6):673–681, 1994.
2. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A Portable Linear Algebra Library for High-Performance Computers. In *Proceedings of Supercomputing '90*, pages 1–10. IEEE Press, 1990.
3. P. V. Bangalore. The Data-Distribution-Independent Approach to Scalable Parallel Libraries. 1995. Master Thesis, Mississippi State University.
4. L. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. In *Proceedings of SIAM Conference on Parallel Processing*, 1997.
5. L. Blackford, J. Choi, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. *ScaLAPACK Users' Guide*. SIAM Press, Philadelphia, PA, 1997.
6. J. Choi. A New Parallel Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers. *Concurrency: Practice and Experience*, 10:655–670, 1998.
7. J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Scientific Programming*, 5:173–184, 1996.
8. J. Choi, J. J. Dongarra, and D. W. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. *Concurrency: Practice and Experience*, 6:543–570, 1994.
9. V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.
10. G. Li and T. F. Coleman. A Parallel Triangular Solver for a Distributed-Memory Multiprocessor. *SIAM J. of Sci. Stat. Computing*, 9:485–502, 1986.
11. W. Lichtenstein and S. L. Johnsson. Block-Cyclic Dense Linear Algebra. *SIAM J. of Sci. Stat. Computing*, 14(6):1259–1288, 1993.
12. R. van de Geijn and J. Watts. SUMMA Scalable Universal Matrix Multiplication Algorithm. LAPACK Working Note 99, Technical Report CS-95-286, University of Tennessee, 1995.
13. R. A. van de Geijn. *Using PLAPACK*. The MIT Press, Cambridge, 1997.